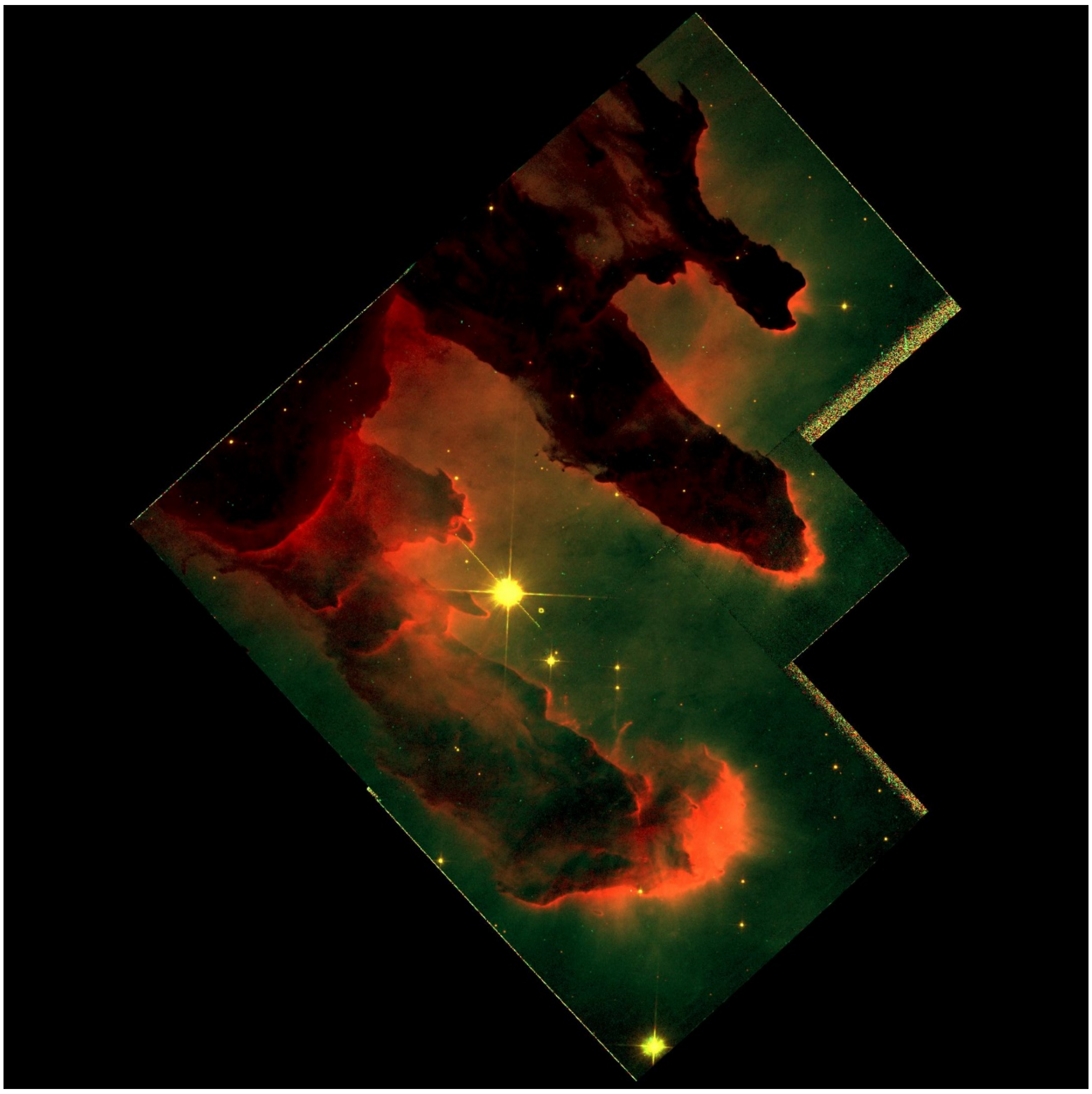
Imaging the Universe

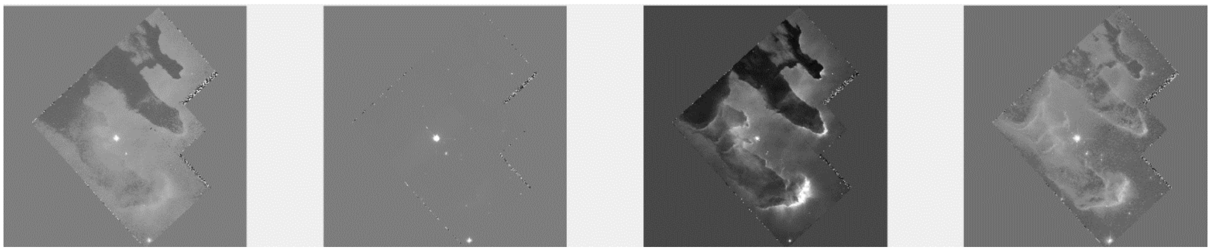
Processing images from the Hubble telescope



By Peter Bier p.bier@auckland.ac.nz

Introduction

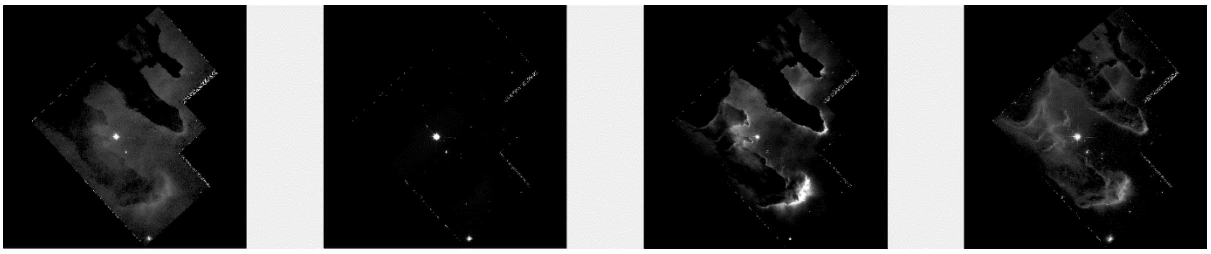
This project centres on the image processing of a set of raw image data (greyscale images) from the Hubble telescope. An example of a set of images is shown below.



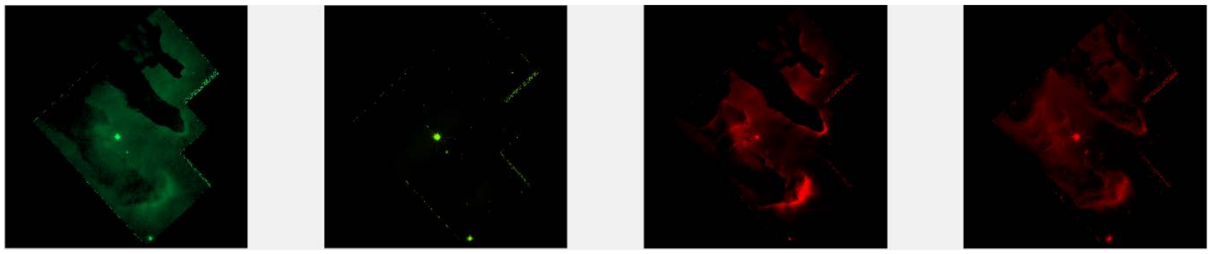
Each image represents the same region of space but has been taken using a different light filter (the filters allow Hubble to record the photons emitted for a particular band of light, centred on a specified wave length).

Some of the images have poor contrast so we will "normalise" them to make full use of the uint8 range of values (0 to 255). After normalising an image the specified background colour will be set to 0 and the peak brightness will be set to 255.

An example of normalising the above images is shown below. The exact result obtained depends on what values were selected to represent the background (dark value) and peak (bright value) for each image.

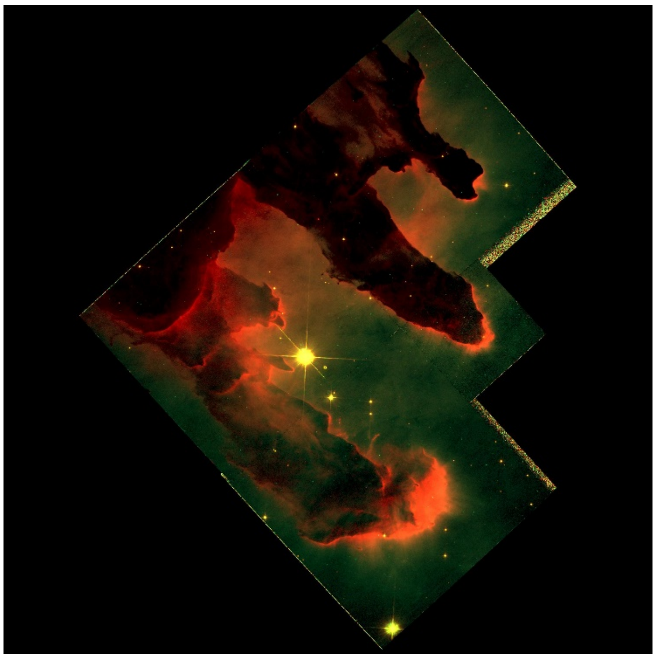


Each image represents a colour band, centred on a particular wavelength. We can colour each greyscale image using either true colour (corresponding to the wave length of light at the centre of the band that was used to take the image) or a user specified colour.



Note that not all images will benefit from using true colour. For example some wave lengths are outside the range of human vision (roughly speaking the visible spectrum corresponds to wave lengths in the 400 to 700 nanometre range). We can still visualise features outside the visible range by manually assigning a visible colour to any images that used filters outside the visible spectrum. We can also observe features related to a particular wavelength more distinctly if we colour wave lengths that are close together using very different colours.

The separate coloured images can be combined into a single image representing a region of space. Here is a composite created from the four separate colour images from the previous page.



If you want a significant challenge, try and write some code that will automatically rotate and crop this combined image (see the images on the next page for some examples of rotated and cropped images).

How to tackle the project

Do not be daunted by the length of this document. It is long because it includes useful background information, tips and a lot of explanation as to exactly what each function needs to do.

The best way to tackle a big programming task is to break it down into small manageable chunks, which has been done for you in the form of eight functions to write. Each function has a detailed description of what it needs to do. Have a read through the entire document and then pick a function to start on. Remember you don't have to start with task 1, although it is a fairly straight forward function to write, so it might be a good warm up. Do as many of the tasks as you can.

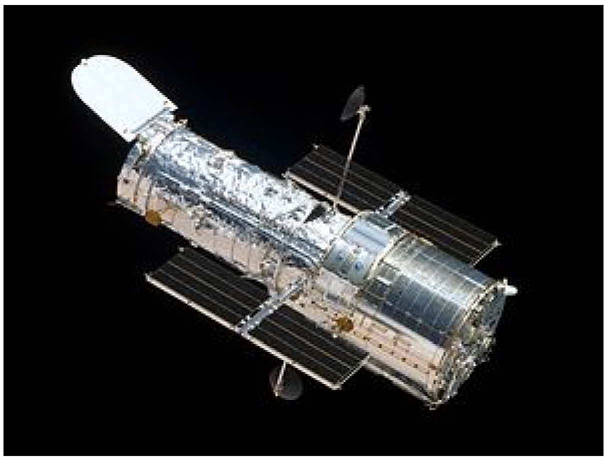
Several functions require careful thought (remember those 5 steps of problem solving!). If you are having trouble understanding how a function should work, remember to work through the problem by hand using some simple data.

Note that I don't necessarily expect everyone to get through all eight functions. Some of them are relatively easy while others are quite tricky. You can still get a pretty good mark even if you don't complete all the functions.

An "A" grade student should be able to nut out all eight functions, B and C grade students might not get a fully working solution. Even if you only get half the functions working, you can still get over 50% for the project, as long as the code you submit is well written (since you get marks for using good style).

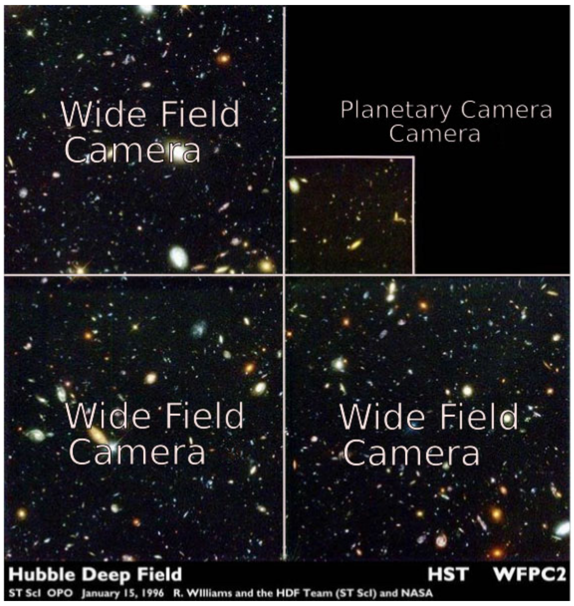
Background on the Hubble Telescope

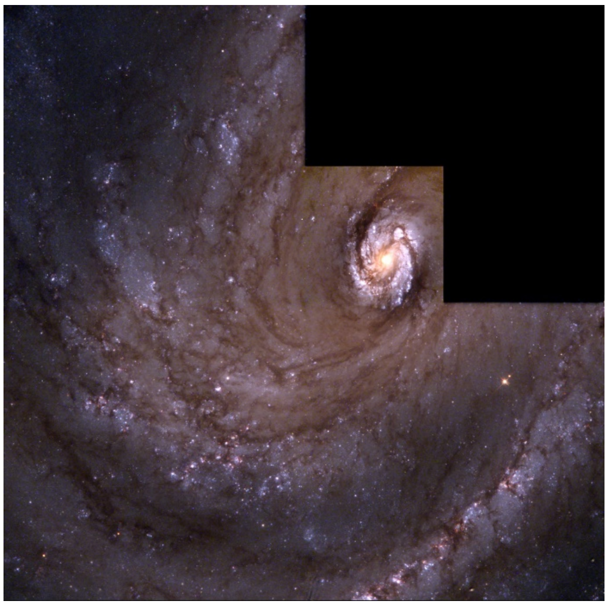
The Hubble telescope is one of the largest and most versatile space telescopes. It is positioned in low earth orbit and can take greyscale images of sections of space, recording the amount of light emitted for a specified wave length.



We will be dealing with images taken with the Wide Field and Planetary Camera (WFPC). This camera takes images from four different quadrants (three wide field and one planetary). The planetary image is higher resolution but doesn't cover as much space, so needs to be scaled down when assembling the quadrants into a single large image.

This gives Hubble WFPC images a characteristic stair step shape, as shown below (*note that both images shown below have been colourised by combining several larger images of different wave lengths of light into a single image*)





Images above are public domain and sourced from:

https://en.wikipedia.org/wiki/Wide_Field_and_Planetary_Camera_2#/media/File:Jfader_hubble.jpg

https://en.wikipedia.org/wiki/Wide_Field_and_Planetary_Camera_2#/media/File:1994-02-b-full_jpg.jpg

The WFPC does not capture images in colour but it can be precisely controlled using filters to capture a band of light centred on a particular wave length (e.g. only green light). It can also capture light waves beyond the visible spectrum including ultraviolet and infrared.

A section of space will be imaged many times, using different filters so that we can learn what amounts of energy are being transmitted by different wave lengths of light. Using this idea we can record how much green light is emitted for a given section of space (or for that matter how much purple light or yellow light or light of any other spectrum colour we might be interested in, including wave lengths beyond the visible spectrum).

Once these filtered images are collected astronomers visualise the results by assigning a visible colour to each filter image and combining the colourised images into a single image.

Note that this combined image may not represent the true colours of that section of space (e.g colour values may have been assigned to invisible parts of the light spectrum so that we can visualise the invisible!).

For a more detailed background on producing images from Hubble data see:

http://hubblesite.org/gallery/behind_the_pictures/

Overview of Functions to Write

1) `DisplayCellImages` displays the images that are stored in a 1D cell array on a single figure, all in one row, using subplots. It is called several times by the `ProcessHubbleImages` script so that we can see the images we are dealing with.

2) `EstimateBackgroundValue` finds the most common value in a greyscale image (where the image is stored in a 2D array of type uint8.) The most common value is a reasonable estimate of the background value to be used when normalising a given image.

3) `NormaliseImage` takes image intensity values and scales them so that the specified background value becomes 0 and the specified peak value becomes 255. This improves the contrast of the image.

4) `ExtractWaveLengthFromFilename` uses the naming conventions of Hubble data files to determine the wave length value of the filter that was used to create the image contained in the file.

5) `WaveLengthToRGB` calculates an approximation of the red, green and blue values that correspond to a specified wave length of light (measured in nanometers).

6) `SpectrumBar` creates an RGB image of a colour spectrum bar. It will use the `WaveLengthToRGB` function to construct the image. It is used by the supplied `PickSpectrumColour` function.

7) `ColourImage` takes a greyscale image and creates an RGB colour image from it using a specified colour to tint the pixels.

8) `CombineImages` takes a set of colour images of different tints and combines them into a single RGB image.

Bonus functions to write (worth bonus marks)

9) `Autorotate` (superhard!!) takes a colour RGB image created from an image set from the Hubble WFPC and automatically rotates it to the traditional orientation.

10) `Autocrop` (a little tricky) takes a nicely aligned Hubble WFPC colour RGB image from the Hubble WFPC and automatically crops it.

Overview of Supplied Code

1) `ProcessHubbleImages` is the main script file for the project. Once you have implemented all the required functions, running this script file will generate a combined image from a set of raw images. **You should read through this script carefully to see how it works.** It calls most of the functions you will be writing.

2) `GetTifFiles` gets the filenames for any tif (true image format) files contained within a specified directory.

3) `PickBackgroundAndPeakValues` lets the user manually pick a dark spot to use as the background value and a bright spot to use as a peak value.

4) `PickSpectrumColour` lets the user select a colour from a colour bar spectrum. It calls the `SpectrumBar` function you write to set up the spectrum bar.

5) `ginputc` is a customised version of the Matlab built-in `ginput` command, which lets the user select points on a figure using coloured crosshairs. It is used by `PickBackgroundAndPeakValues`

Overview of Supplied Images

To get you started I have supplied four images of the Eagle Nebula sourced from the Hubble Legacy Archive. These images have been exported as uint8 tif images and in some cases have had filters applied to improve contrast.

If you would like to download some more images, see the appendix on **Sourcing Hubble Images** for instructions on how to do so.

The images are quite large so you may wish to scale them down if you have a slower machine and are finding that execution time becomes an issue for you when you are developing your code.

Task 1: Write the `DisplayCellImages` function

The purpose of this function is to display the images that are stored in a 1D cell array on a single figure, all in one row, using subplots. Each subplot is titled with a message followed by a space and an image number. This function is called several times by the `ProcessHubbleImages` script so that we can see the images we are dealing with.

Inputs (in order):

1) A 1D **cell array** containing one or more images. The images could be either greyscale images stored in 2D arrays (of type uint8) or RGB colour images stored in 3D arrays (of type uint8).
2) A string containing a message to use at the start of each subplot title (each subplot will be titled using this message followed by a space and then an image number).
3) A number to use when creating the figure (i.e. a value to pass in when calling the figure command).

Output:

NONE (the function displays a figure so does not need to return any values).

Example:

Here is an example of some calls to `DisplayCellImages`

```
% make three grey images of different shades and assign
% to cell array
grey{1} = ones(100,'uint8')*64;
grey{2} = ones(100,'uint8')*128;
grey{3} = ones(100,'uint8')*192;
DisplayCellImages(grey,'Grey image',1)

% make two random colour images and assign to cell array
colour{1} = uint8(255*rand(100,100,3));
colour{2} = uint8(255*rand(100,100,3));
DisplayCellImages(colour,'Random colour',2)
```

Running the above script would produce the figures shown overleaf.

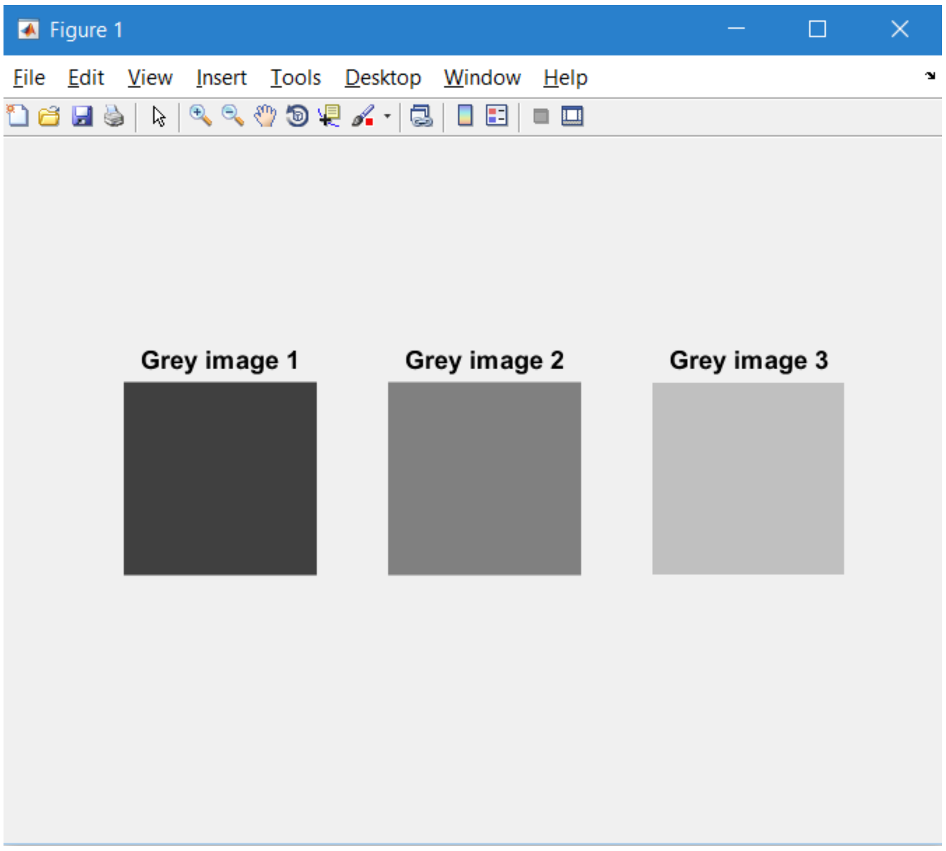


Figure 1
File Edit View Insert Tools Desktop Window Help
Grey image 1
Grey image 2
Grey image 3

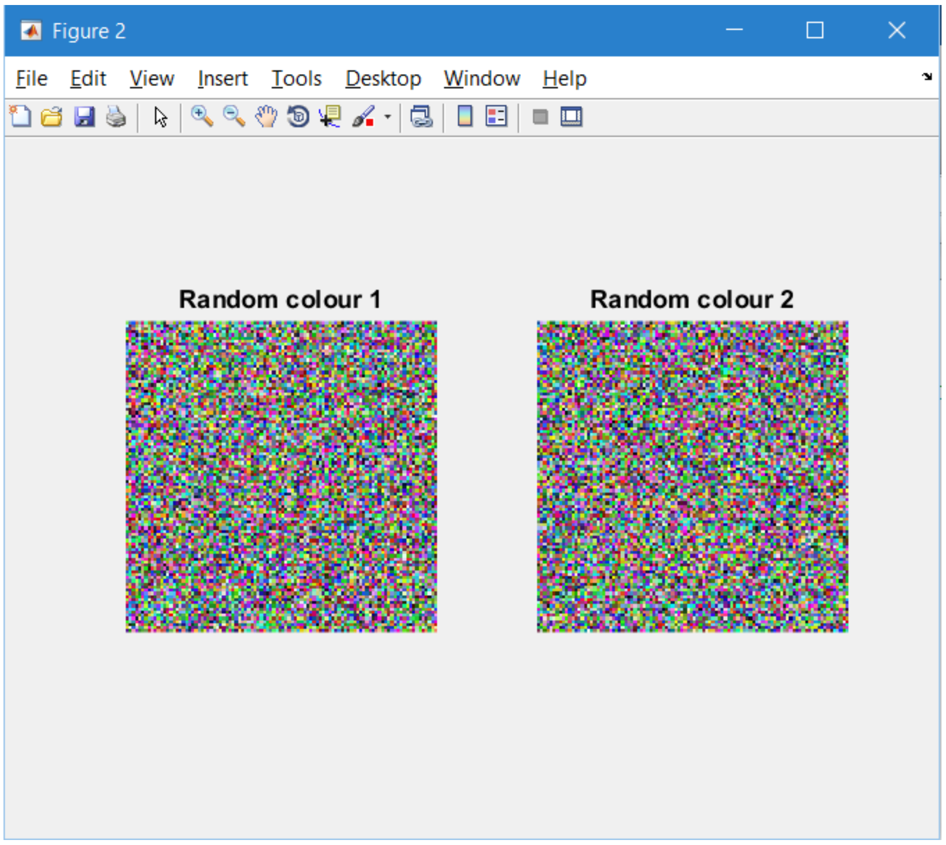


Figure 2
File Edit View Insert Tools Desktop Window Help
Random colour 1
Random colour 2

Task 2: Write the `EstimateBackgroundValue` function

The purpose of this function is to provide an estimate of the background value for an image. It works by finding the most common intensity for a greyscale image (i.e. the most common integer in a 2D array of uint8 values).

Input

A 2D array of uint8 values representing a greyscale image.

Output

The most common value in the array (returned as a unit8). If there are several values that are the most common (because they occur the same number of times) then the lowest of these is returned.

Tip

There are many ways to solve this problem. One possible approach is to loop through all elements of the array and keep a running count of how many times each value occurs. The most common value will then be the one with the largest count. If you decide to use this approach I recommend working through the problem by hand on several very small arrays before attempting to code it.

Example

Here is an example of some calls to `EstimateBackgroundValue,` along with the expected return values.

```
% set up a small 2x4 array of unit8 values
A =uint8( [128 255 128  0; 0 0 255 64] );
a = EstimateBackgroundValue(A) % will return 0

% set up a small 3x4 array of unit8 values
B =uint8( [3 2 1; 0 0 0; 1 2 3; 1 1 3] );
b = EstimateBackgroundValue(B) % will return 1

% set up a 3x6 array of unit8 values
C = zeros(3,6,'uint8');
C(1,:) = 100; % set row 1 to 100
C(2,:) = 50; % set row 2 to 50
C(3,:) = 200; % set row 3 to 200
c = EstimateBackgroundValue(C) % will return 50
```

Task 3: Write the `NormaliseImage` function

The purpose of this function is to improve the image contrast by normalising the greyscale image, to take advantage of the full range of values that can be represented by uint8 variables (i.e. 0 to 255). The image is linearly scaled to move a specified background value (dark value) to 0 and the peak value (bright value) to 255. The peak value is assumed to be larger than the background value.

For more information on normalisation see:
https://en.wikipedia.org/wiki/Normalization_(image_processing)

Inputs (in order)

1) A greyscale image to normalise (stored as a 2D array of unsigned 8 bit integers).
2) A "background" value (that will become the darkest value). This will be an unsigned 8 bit integer value in the range 0 to 255 inclusive.
3) A "peak" value (that will become the brightest value). This will be an unsigned 8 bit integer value in the range 0 to 255 inclusive and is assumed to be larger than the background value.

Output

A 2D array of unsigned 8 bit integers containing the normalised greyscale image.

Calculation

The equation for normalising a pixel intensity value, *v*, is as follows:

$$n = 255\frac{(v-b)}{(p-b)}$$

where *p* is the peak value and *b* is the background value. This transforms background values to zero and peak values to 255 (with linear interpolation for values in between).

Note that calculating a normalised pixel intensity may result in a floating point number, in which case the resulting value should be **rounded** to the nearest integer.

Using the above process may result in values outside of the range 0 to 255. For values outside this range:

- Any values less than 0 should be assigned a final of value 0.
- Any values greater than 255 should be assigned a final value of 255.

VERY IMPORTANT TIP

Inside your function you may want to convert your unsigned 8 bit integer values to doubles (using the Matlab `double` function) before calculating values. This is an important step since the largest value an unsigned 8 bit integer can represent is 255 and parts of your calculation may exceed this value. See the appendix WARNING about working with uint8 values for more details.

Example

Here is an example of some calls to `NormaliseImage`

```
% set up a small 2x4 array of unit8 values
A = uint8( [128 255 128  0; 0 0 255 64] );
NA = NormaliseImage(A,0,255) % no change

% set up a small 3x4 array of unit8 values
B = uint8( [3 2 1; 0 0 0; 1 2 3; 1 1 3] );
NB1 = NormaliseImage(B,0,3)
NB2 = NormaliseImage(B,1,3)
NB3 = NormaliseImage(B,2,3)

% set up a 3x6 array of unit8 values
C = zeros(3,6,'uint8');
C(1,:) = 100; % set row 1 to 100
C(2,:) = 50; % set row 2 to 50
C(3,:) = 200; % set row 3 to 200
NC1 = NormaliseImage(C,50,200)
NC2 = NormaliseImage(C,50,100)
```

Here is the result of running the above:

```
NA =

  128  255  128    0
    0    0  255   64

NB1 =

  255  170   85
    0    0    0
   85  170  255
   85   85  255

NB2 =

  255  128    0
    0    0    0
    0  128  255
    0    0  255
```

```
NB3 =

  255    0    0
    0    0    0
    0    0  255
    0    0  255


NC1 =

   85   85   85   85   85   85
    0    0    0    0    0    0
  255  255  255  255  255  255


NC2 =

  255  255  255  255  255  255
    0    0    0    0    0    0
  255  255  255  255  255  255
```

Task 4: Write the `ExtractWaveLengthFromFilename` function

The purpose of this function is to extract the wave length (in nanometres) used as the centre of the filter band for the image from the filename. Hubble images follow a naming convention for their files. The text identifying the instrument (e.g. wfpc2) is followed by an underscore and then an f for filter) plus a 3 digit number specifying the wave length that was used for the filter and then either the letter n (for narrow), the letter m (for medium) or the letter w (for wide band).

For example, the file named `'hst_05237_02_wfpc2_f656n_wf_sci.tif'` was created using WFPC2 and a wave length of 656 nanometres.

Your code should extract the three digit value that is sandwiched between the characters `'_f'` and the characters `'n_'`, `'m_'`, or `'w_'` (depending on whether it was a narrow, medium or wideband filter).

It should NOT rely on the name of the instrument being wfpc2 as you may wish to use images taken by Hubble using other instruments.

Input

A string containing the filename of an image.

Output

The wave length associated with the image, returned as a double. This will be the three digit value that is sandwiched between the characters `'_f'` and `'n_'` (if a narrowband filter was used) or the `'_f'` and `'m_'` (if a mediumband filter was used).or the `'_f'` and `'w_'` (if a wideband filter was used). If a wave length could not be extracted because the file name does not follow the naming convention a value of -1 should be returned.

Example

Here is an example of some calls to `ExtractWaveLengthFromFilename`

```
f1 = 'hst_05773_05_wfpc2_f502n_wf_sci.tif'
w = ExtractWaveLengthFromFilename(f1) % will return 502

f2 = 'hst_05773_05_wfpc2_f555w_wf_sci.tif'
w = ExtractWaveLengthFromFilename(f2) % will return 555

f3 = 'hlsp_clash_hst_acs-30mas_a383_f625w_v1_drz.tif'
w = ExtractWaveLengthFromFilename(f3) % will return 625

f4 = 'image.tif'
w = ExtractWaveLengthFromFilename(f4) % will return -1

f5 = 'hst_05773_05_wfpc2_f547m_wf_sci.tif'
w = ExtractWaveLengthFromFilename(f5) % will return 547
```

Task 5: Write the `WaveLengthToRGB` function

The purpose of this function is to convert a wave length (measured in nanometres) into an equivalent set of RGB values that approximate the colour of light for the specified wave length. *It should be noted that this is an approximation only.*

Input

A wave length (measured in nanometres), e.g. the value 400.

Outputs (in order)

1) A red intensity value (in the range 0 to 255) for the specified wave length, returned as an unsigned 8 bit integer (uint8).
2) A green intensity value (in the range 0 to 255) for the specified wave length, returned as an unsigned 8 bit integer (uint8).
3) A blue intensity value (in the range 0 to 255) for the specified wave length, returned as an unsigned 8 bit integer (uint8).

Colour calculation

The table below specifies what formulae to use as multipliers for wave lengths within different ranges.

Colour	Wave length Range	Band Width	R value multiplier	G value multiplier	B value multiplier
Violet – Blue	380<=w<=440	60	(440-w)/60	0	1
Blue – Cyan	440<w<=490	50	0	(w-440)/50	1
Cyan – Green	490<w<=510	20	0	1	(510-w)/20
Green – Yellow	510<w<=580	70	(w-510)/70	1	0
Yellow- Red	580<w<=645	65	1	(645-w)/65	0
Red	645<w<=780	135	1	0	0

Once we have identified the wave length range, we can calculate the values of R, G and B by multiplying 255 by the multipliers specified in the corresponding row of the table (rounding to the nearest integer if needed). Any wave length not in the range 380<=w<=780 will have the values R=0, G=0, B=0 returned.

Here are a few examples of calculating the RGB values for some wave lengths.

A wave length of 400 is in the “Violet – Blue” range. This means we have an R value multiplier specified by (440-w)/60. This results in $R = 255 \times \frac{440-400}{60} = 170$.

Our G value multiplier is 0, so $G = 255 \times 0 = 0$.

Our B value multiplier is 1, so $B = 255 \times 1 = 255$.

Hence we return the values *R*=170, *G*=0 and *B*=255 to represent this wave length.

A wave length of 500 is in the "Cyan – Green" range. This means we have an R value multiplier of 0, so $R = 255 \times 0 = 0$.

Our G value multiplier is 1, so $G = 255 \times 1 = 255$.

Our B value multiplier is specified by (510-w)/20, this results in $B = \frac{510-500}{20} \times 255 = 127.5$ which rounds to 128.

Hence we return the values *R*=0, *G*=255 and *B*=128 to represent this wavelength.

Example

Here is an example of some calls to `WaveLengthToRGB`

```
[R,G,B]=WaveLengthToRGB(400) % gives 170,0,255
[R,G,B]=WaveLengthToRGB(500) % gives 0,255,128
[R,G,B]=WaveLengthToRGB(580) % gives 255,255,0
[R,G,B]=WaveLengthToRGB(800) % gives 0,0,0
```

Task 6: Write the `SpectrumBar` function

The purpose of this function is to create an image of a spectrum bar, 100 pixels (rows) in height, covering the specified wave length range (from a low wave length up to a high wave length).

Inputs (in order)

1) A low wave length value (this will be used to colour the pixels in the first column of the image).
2) A high wave length value (this will be used to colour the pixels in the last column of the image).

Output

A 3D array of unsigned integers representing a spectrum bar that spans the wave length range. Each successive column of pixels corresponds to the colour of a wavelength 1 nanometre higher than the previous one. The array will have 100 rows, n columns and 3 layers where the number of columns is given by $n = h_w - l_w + 1$, where h_w is the highest wave length and l_w is the lowest wave length.

This function will use your `WaveLengthToRGB` function to fine the appropriate RGB values for each wave length.

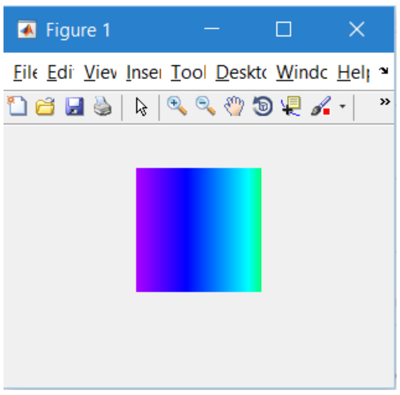
Example

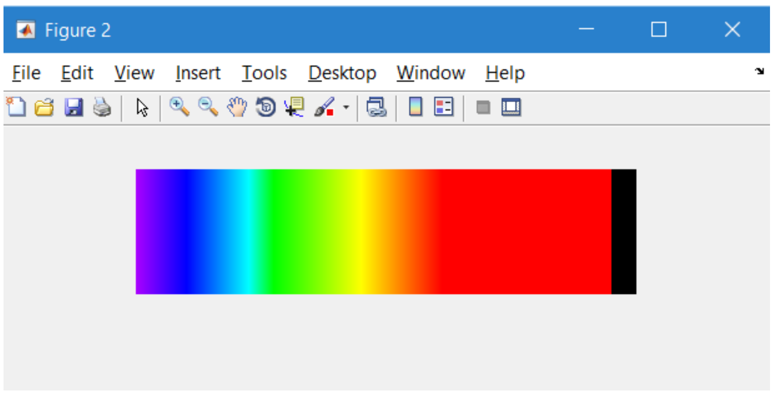
Here is an example of some calls to `SpectrumBar`

```
S=SpectrumBar(400,500) % will produce a 100x101x3 array
figure(1)
imshow(S)

S=SpectrumBar(400,800) % will produce a 100x401x3 array
figure(2)
imshow(S)
```

Below are the spectrums produced by this code.





Task 7: Write the `ColourImage` function

The purpose of this function is to colour a greyscale image using a specified RGB colour to tint the pixels.

Inputs (in order)

1) A 2D array of unsigned 8 bit integers representing a greyscale image.
2) A red intensity value, r_{in} (in the range 0 to 255), stored as an unsigned 8 bit integer.
3) A green intensity value, g_{in} (in the range 0 to 255), stored as an unsigned 8 bit integer.
4) A blue intensity value, b_{in} (in the range 0 to 255), stored as an unsigned 8 bit integer.

Output

A 3D array of unsigned 8 bit integers representing a tinted colour image. The image will have the same number of rows and columns as the 2D image passed in as the first input value. It will have 3 layers, to represent the different values of R, G and B for each pixel.

Calculation

The equations for calculating the amount of red, green and blue corresponding to a greyscale pixel of intensity value, *v*, are as follows:

$$R = \frac{r_{in}v}{255} \qquad G = \frac{g_{in}v}{255} \qquad R = \frac{b_{in}v}{255}$$

If an equation gives a floating point value, the result should be **rounded** to the nearest integer.

Tip

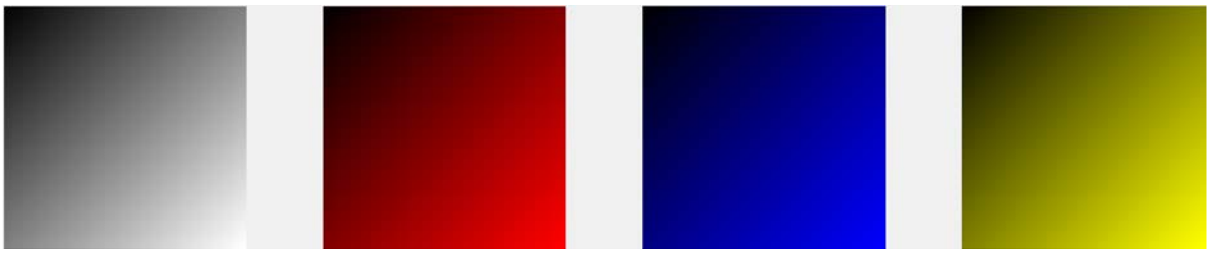
Inside your function you may want to convert your unsigned 8 bit integer values to doubles (using the Matlab `double` function) before calculating values. This is an important step since the largest value an unsigned 8 bit integer can represent is 255 and parts of your calculations may exceed this value. See the appendix WARNING about working with uint8 values for more details.

Example

Here is an example of some calls to `ColourImage`

```
% create a greyscale image
G = zeros(128,'uint8');
for i=1:128
    for j=1:128
        G(i,j)=i+j;
    end
end
C1=ColourImage(G,255,0,0); % colour it with pure red
C2=ColourImage(G,0,0,255); % colour it with pure blue
C3=ColourImage(G,255,255,0); % colour it with yellow
figure(1)
subplot(1,4,1)
imshow(G)
subplot(1,4,2)
imshow(C1)
subplot(1,4,3)
imshow(C2)
subplot(1,4,4)
imshow(C3)
```

Here are the resulting images displayed



Task 8: Write the `CombineImages` function

The purpose of this function is to combine a set of coloured images of varying tints (stored in a 1D cell array) into a single RGB image that represents a visualisation of a region of space.

Input

A 1D **cell array** containing 1 or more colour images (each image will be a 3D array of unsigned 8 bit integers). It is assumed that all images will have the same size.

Output

A 3D array of unsigned 8 bit integers representing the combined image.

Calculation

There are many possible ways to combine images. We will implement the simplest option of adding corresponding elements together from each of the input images.

For example the red intensity value for the pixel in row 1, column 1 of the combined image will be the sum of the red intensity values of each pixel in row 1, column 1 of the input images.

If the sum comes to more than 255 when calculating an intensity for the combined image, a value of 255 will be assigned.

Example

Here is an example of a call to `CombineImages` with, showing the resulting purple image.

```
% create a greyscale image
G = zeros(128,'uint8');
for i=1:128
    for j=1:128
        G(i,j)=i+j;
    end
end
C{1}=ColourImage(G,255,0,0); % colour it with pure red
C{2}=ColourImage(G,0,0,255); % colour it with pure blue
T=CombineImages(C) % should produce a purple image
figure(2)
imshow(T)
```

BONUS Task 9: Write the `Autorotate` function

The purpose of this function is to automatically align a colour image that has been created from images obtained from the WFPC. Writing the function is OPTIONAL but will be worth a small number of bonus marks. WARNING – This is super hard (unless you use the image toolbox, in which case it is still moderately hard!).

Input

A 3D array of unsigned 8 bit integers representing a misaligned image (where the edges of the region of interest are NOT parallel to the edges of the image).

Output

A 3D array of unsigned 8 bit integers representing an aligned image (where the edges of the region of interest are parallel to the edges of the image.

BONUS Task 10: Write the `Autocrop` function

The purpose of this function is to automatically crop a nicely **aligned** image that has been created from images taken using the WFPC. Writing the function is OPTIONAL but will be worth a small number of bonus marks. WARNING – This is a little tricky.

Input

A 3D array of unsigned 8 bit integers representing a nicely aligned image (where the edges of the region of interest are parallel to the edges of the image).

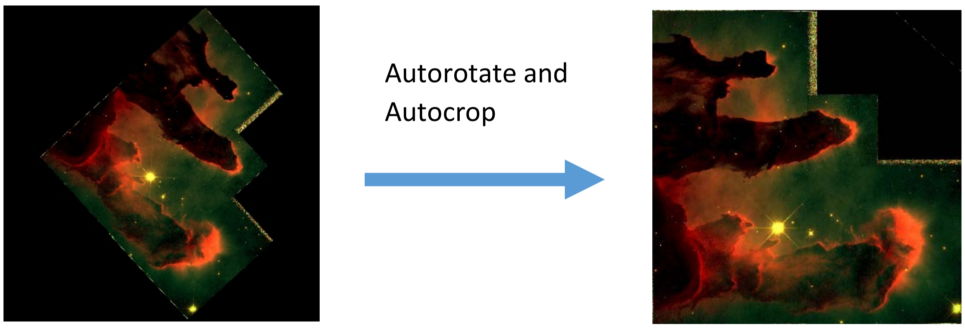
Output

A 3D array of unsigned 8 bit integers where the regions around the image that do not show any sections of space have been removed.

Example

Here is an example of a call to `Autorotate` and `Autocrop`

```
R=AutoRotate(C);
F=AutoCrop(R)
```



How the project is marked

A mark schedule and test scripts will be published prior to the due date, outlining exactly how marks will be allocated for each part of the project and giving you the opportunity to test your code. You will receive marks both for correctness (does your code work?) and style (is it well written?).

Each of the **eight** required function will be marked independently for correctness, so even if you can't get everything to work, please do submit the functions you have written. Some functions may be harder to write than others, so if you are having difficulty writing a function you might like to try working on a different function for a while and then come back to the harder one later.

Note it is still possible to get marks for good style, even if your code does not work at all! Style includes elements such as using sensible variable names, having header comments, commenting the rest of your code, avoiding code repetition and using correct indentation.

Each function can be written in less than 20 lines of code (not including comments) and some functions can be written using just 3 lines but do not stress if your code is longer. It is perfectly fine if your project solution runs to several hundred lines of code, as long as your code works and uses good programming style.

How the project is submitted

Submission is done by uploading your code to the Aropa website. More information will be provided on how to submit the project to Aropa in a Canvas announcement. If you want to be out of Auckland over the break and still want to work on the project, that is perfectly fine as long as you have access to a computer with Matlab installed and can access the internet to upload your final submission.

Checklist

Here is a list of the **eight** functions specified in this document. Remember to include all eight (or as many of them as you managed to write) in your project submission. The filenames should be EXACTLY the same as shown in this list (including case). They should also work exactly as described in this document (pay close attention to the prescribed inputs and outputs). In addition if your functions call any other "helper" functions that you may have written, remember to include them too.

- `DisplayCellImages`
- `EstimateBackgroundValue`
- `NormaliseImage`
- `ExtractWaveLengthFromFilename`
- `WaveLengthToRGB`
- `SpectrumBar`
- `ColourImage`
- `CombineImages`

Am I allowed to use Tool boxes?

Matlab consists of a core set of functions (which is all you get if you purchased the $59 version online). There are also many optional tool boxes (you will have received some of these if you bought the more expensive $99 bundled version and some tool boxes are available in the labs too).

Tasks one to eight should be completed using the supplied code and **core Matlab functions only** (i.e. you should NOT use the image tool box or any of the other tool boxes).

While some of the tool boxes might prove useful for solving tasks one to eight, the aim of this project is to develop your coding skills by giving you plenty of practice at the basics we have taught you. It is possible to write all of tasks one through eight using only the concepts we have covered so far. *For the bonus tasks you may use tool boxes if you wish.*

Any questions?

If you have any questions regarding the project please first check through this document. If it does not answer your question then feel free to ask the question on the class Piazza forum. Remember that **you should NOT be publicly posting any of your project code** on Piazza, so if your question requires that you include some of your code, make sure that it is posted as a PRIVATE piazza query.

Appendix: Sourcing Hubble Images

Images supplied with this project have been taken from the Hubble Legacy Archive:
http://hla.stsci.edu/

Feel free to download images to experiment with visualising different parts of the galaxy.

The following guide is helpful for stepping you through how to download and access images:

http://www.wikihow.com/Process-Your-Own-Colour-Images-from-Hubble-Data

Note the naming convention used for each file can provide you with useful information. For images taken by the WFPC2 you will see the text wfpc2 third to last in the file name. This is followed by the filter name (e.g. f555w) and then the camera type (wf or pc).

The filters all start with an f followed by a number and then a w (for wide band) or an n (for narrow band). The number tells you the central value for the passband of the filter (measured in nanometres). For example f555w is a wide band filter letting in a band of light centered on 555 nanometres past.

Remember that the visible light spectrum corresponds roughly to wave lengths from 400 to 700 nanometres.

Here are some examples of wideband filters:

F435W, F439W, F450W, F555W, F606W, F675W, F702W, F791W, and F814W.

Here are a few examples of narrowband filters

F437N, F502N, F656N, F658N, and F673N.

Appendix: WARNING about working with uint8 values

When performing mathematical calculations with values stored as unsigned 8 bit integers (uint8), you can run into the issue of your calculations not giving the results you might expect. This is due to the limitations of what values can be stored in an unsigned 8 bit integer. These limitations are a potential source of bugs and these bugs can be REALLY hard to track down.

Consider this example:

```
x = uint8(100) % make a uint8 value

y = x*3/2 % we expect y to be 150 but it contains 128.
```

After running this code you might expect `y` to contain the value 150 but this is NOT the value that gets assigned to the variable `y`. Instead it is assigned a value of 128. The seemingly strange behaviour is a consequence of the order of operations and of the variable `x` being an unsigned 8 bit integer.

As `x` is of type uint8, Matlab assumes the result will be assigned to a variable of the same type (i.e. `y` is assumed to be uint8). The first part of the calculation, `x*3`, will also be interpreted as an unsigned 8 bit value. We know `100*3=300` but 255 is the largest value that can be represented, so `100*3` produces a value of 255. Then we divide this by 2 to get 127.5, which needs to be rounded since uint8 variables can't handle non-integers. Hence we get a value of 128, instead of the expected 150.

To get the mathematically correct result you can convert `x` to a double first (so that all values are treated as doubles). At the end if `y` needs to be an unsigned 8 bit integer it can be converted to one. For example:

```
x = uint8(100) % we start off with a uint8 value

x = double(x); % use the double function to convert x to a double

y = x*3/2 % will now calculate the value of y as a double,

% y has the expected answer of 150.

y=uint8(y) % convert answer to uint8
```

Another alternative is to change the order of the operations so that you don't hit the 255 limit:

```
x = uint8(100) % make a uint8 value

y = x/2*3 % this will work fine since x/2 is 50 and 50*3 is 150
```

TIP: The safest option if doing any calculations with uint8 values that might exceed 255 is to convert values to doubles first, do the calculation and then convert the final result back to uint8, if needed.